Available online at www.sciencedirect.com

SCIENCE DIRECT

Electronic Notes in Theoretical Computer Science

ELSEVIER Electronic Notes in Theoretical Computer Science 110 (2004) 133–148

www.elsevier.com/locate/entcs

Semantics-preserving Migration of Semantic Rules During Left Recursion Removal in Attribute Grammars

Wolfgang Lohmann[a] Günter Riedewald[a] Markus Stoy[a]

[a] *Department of Computer Science*
University of Rostock
18051 Rostock, Germany

Abstract

Several tools for source-to-source transformation are based on top down parsers. This restricts the user to use grammars without left recursion. Removing left recursion of a given grammar often makes it unreadable, preventing a user from concentrating on the original grammar. Additionally, the question arises, whether the tool implements the semantics of the original language, if it is implemented based on a different grammar than in the original language definition. Moreover, existing implementations of semantics for the original grammar cannot be reused directly.
The paper contributes to the field of automatic migration of software (here semantic rules) induced by a grammar change. It revises removal of left recursion in the context of grammar adaptations and demonstrates, that while removing left recursion at the same time the semantic rules can be migrated automatically. Thus, a programmer can continue to use semantic rules on a left recursive grammar. The problem is explained and justified.

Keywords: grammar engineering, grammar adaptation, attribute grammar, migration of semantic rules, transformation, parsing

1 Introduction

In the paper we consider the consequences of left recursion removal to semantics associated with grammar rules. Our starting point is the need for grammar engineering after semantic rules have been written for the grammar already. We will demonstrate that during automatic left recursion removal in attribute grammars semantic rules can be migrated automatically.

Grammar engineering Work with grammars is present in software development as well as in maintenance. Grammars are used to describe structure of

doi:10.1016/j.entcs.2004.06.006

data, to derive tools for manipulating those data, or to serve as reference between developers, e.g. a language definition. As other software artifacts, grammars are subject to change, e.g. adaptations to make it usable for parser generation, evolution of grammars (grammar corrections, changes and extensions of the language), grammar recovery from existing tools or documents, and refactoring of grammars (to make them more readable, parts better reusable, e.g. for tools adaptable to several language dialects).

Need for left recursion removal Removal of left recursion in grammars is an adaptation of the grammar to fit technical demands. Many syntactical structures are expressed naturally using recursion, often both, left and right recursion. However, there are tools like ANTLR, JavaCC, TXL, Prolog-based tools, dealing somehow with recursive descent parser generation, for the ease of combination with semantics [21], which would fall into infinite recursion. Removal of left recursion is known in compiler construction for over 40 years, and mostly considered wrt. to context-free grammars or to development of compilers. However, necessity of left recursion removal arises not only in compiler construction, but also during language development, prototyping, and in software maintenance, especially for adaptations in already used and tested grammars.

Technical challenge The first problem is that removal of left recursion leads to a badly readable grammar. More elaborate semantic rules are necessary. However, the user wants to work on the most comprehensible grammar, or even the reference grammar, if possible. Often the grammar is rewritten using EBNF, where left recursion turns into iteration, which might result in a problem with semantics in loops. Next, a language definition consists of syntax and semantics definitions. If the syntax is modified due to technical demands, this leads to changed semantics. Is the meaning of a language construct unchanged? Finally, if there are semantic rules for the left recursive grammar already (e.g. given as logic language), how are they affected by the change? Can they be reused or have all to be discarded?

Results and benefits We argue the semantic meaning associated to grammar symbols of the original grammar can be still reconstructed after automatic left recursion removal. This will be justified for S-attributed grammar. We will discuss, how the approach can be generalised for multi-pass attribute grammars. Programmers benefit from our approach, because they can now work on a grammar similar to the reference grammar, i.e. one possibly containing left recursion. The adaptation of the grammar and the semantic rules can be done automatically, and can be implemented as a preprocessor, as shown in Figure 1. The approach can be combined with the above mentioned tools (e.g. ANTLR, Prolog-based tools).

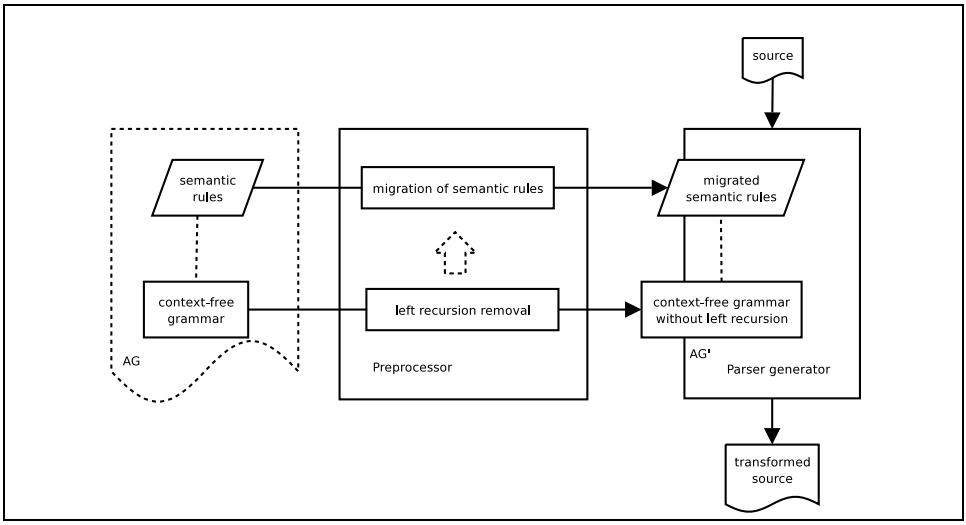

Fig. 1. An example use of the approach

Remainder of the paper

Section 2 recalls the notions of attribute grammars. Section 3 uses the small example of arithmetic expressions to explain the basic idea for the transformation of semantic rules in Section 4. Section 5 gives a justification. Treatment of other kinds of attribute grammars is disussed in Section 6. Section 7 reports on practical experience so far. Section 8 points to some related work, before the paper is summarised in Section 9.

2 Notions of Attribute Grammars

This section recalls the definition of attribute grammars (AG). The following formal definition is similar to [1]. Semantic conditions, which can restrict the language generated by the context-free grammar are omitted without loss of generality of our approach. The semantics is as given in [1]. For the origin of attribute grammars, the reader is referred to Irons [6], and Knuth [9].

An attribute grammar without semantic conditions is a four-tuple $AG = (G, SD, AD, R)$, where

(i) $G = (V_N, V_T, P, S)$ is the base context-free grammar. V_N and V_T are sets of nonterminals and terminals, $V = V_N \cup V_T$ and $V_N \cap V_T = \emptyset$. P is a finite set of production rules, $S \in V_N$ denotes the start symbol, $p \in P$ will be written as $p : X_0^p \to X_1^p \dots X_{n_p}^p$, where $n_p \geq 0$, $X_0^p \in V_N$ and $X_k^p \in V$ for $1 \leq k \leq n_p$.

(ii) $SD = (TYPES, FUNCS)$ denotes the semantic domain. $TYPES$ is a

finite set and $FUNCS$ a finite set of total functions with $type_1 \times \ldots \times type_n \rightarrow type_0$, $n \geq 0$ and $type_i \in TYPES$ $(0 \leq i \leq n)$.

(iii) $AD = (A_I, A_S, TYPE)$ denotes the attributes. Each symbol $X \in V$ gets finite sets of synthesized and inherited attributes associated, $A_I(X)$ and $A_S(X)$. $A(X) = A_I(X) \cup A_S(X)$ and $A_I(X) \cap A_S(X) = \emptyset$, and $A = \cup_{X \in V} A(X)$ (for A_I and A_S analogously). An attribute a of some symbol X can be written $X.a$, if necessary for distinguishing. For $a \in A$ $TYPE(a) \in TYPES$ is the set of values of a ($TYPE = \cup_{a \in A} TYPE(a)$).

(iv) $R = \cup_{p \in P} R(p)$ denotes the finite set of semantic rules associated with a production $p \in P$. The production $p : X_0^p \rightarrow X_1^p \ldots X_{n_p}^p$ has an attribute occurrence $X_k^p.a$, if $a \in A(X_k^p)$. The set of all attribute occurrences of a production p is written as $AO(p)$. It can be divided into two disjoint subsets of defined occurrences $DO(P)$ and used occurrences $UO(p)$, which are defined as follows:

$$DO(p) = \{X_0^p.s \mid s \in A_S(X_0^p)\} \cup \{X_k^p.i \mid i \in A_I(X_k^p) \wedge 1 \leq k \leq n_p\},$$
$$UO(p) = \{X_0^p.i \mid i \in A_I(X_0^p)\} \cup \{X_k^p.s \mid s \in A_S(X_k^p) \wedge 1 \leq k \leq n_p\}.$$

The semantic rules of $R(p)$ define, how values of attribute occurrences in $DO(p)$ can be computed as function of other attribute occurrences of $AO(p)$. The defining rule for attribute occurrence $X_k^p.a$ is of the form

$$X_k^p.a := f_{ka}^p(X_{k_1}^p.a_1, \ldots, X_{k_m}^p.a_m)$$

where $X_k^p.a \in DO(p)$, $f_{ka}^p : TYPE(a_1) \times \ldots \times TYPE(a_m) \rightarrow TYPE(a)$, $f_{ka}^p \in FUNCS$ and $X_{k_i}^p \in AO(p)$ for $1 \leq i \leq m$. The occurrence of $X_k^p.a$ depends on $X_{k_i}^p.a_i$ $(1 \leq i \leq m)$. An AG is in normal form, if for each semantic rule additionally holds: $X_{k_i}^p.a_i \in UO(p)$. Each AG can be transformed into normal form. Without loss of generality, we assume our grammar to be in normal form.

There are several subclasses of AG, among those S-attributed grammars (S-AG) and I-attributed grammars (I-AG). For S-AG is $A_I = \emptyset$ and the computation is done bottom up, i.g. the attributes of the root node contain the determined meaning of the program. Analogously, for I-AG is $A_S = \emptyset$, and the computation is top-down. The meaning is in the leaves.

3 Left recursion removal in an example AG

We demonstrate the basic idea using the common simple example for left recursive definition of arithmetic expressions. The context-free part of the grammar implements priority of arithmetic operators. Figure 2 gives the general algorithm for left recursion removal for context-free grammars (cf. e.g. [18]). This algorithm has to be extended to deal with semantic rules,

Input: $G = (V_N, V_T, P, S)$ without ϵ-productions and cycles;
without loss of generality let $V_N = \{A^1, \ldots, A^N\}$
Output: $G' = (V'_N, V_T, P', S)$ without left recursion
for $i := 1$ to N **do**
{Removal of indirect left recursion}
for $j := 1$ to $i - 1$ **do**
replace productions of pattern $A^i \to A^j\beta$
by $A^i \to \alpha_1\beta \mid \ldots \mid \alpha_k\beta$,
where $A^j \to \alpha_1 \mid \ldots \mid \alpha_k$ are the current productions of A^j
end for
{Removal of direct left recursion}
replace productions of pattern $A^i \to A^i\alpha_1 \mid \ldots \mid A^i\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$
where no β_k starts with A^i
by $A^i \to \beta_1 A^{i'} \mid \ldots \mid \beta_m A^{i'}$ and $A^{i'} \to \alpha_1 A^{i'} \mid \ldots \mid \alpha_n A^{i'} \mid \epsilon$
where $A^{i'}$ is a new introduced nonterminal
end for

Fig. 2. Left recursion removal for context-free grammars

			$E \to TE'$		$\{\ ?\ \}$
$E_0 \to E_1 + T$		$\{\ E_0.v := E_1.v + T.v\ \}$	$E' \to +TE'$		$\{\ ?\ \}$
$\mid$	$E_1 - T$	$\{\ E_0.v := E_1.v - T.v\ \}$	$\mid$	$-TE'$	$\{\ ?\ \}$
$\mid$	T	$\{\ E_0.v := T.v\ \}$	$\mid$	ϵ	$\{\ ?\ \}$
$T_0 \to T_1 * F$		$\{\ T_0.v := T_1.v * F.v\ \}$	$T \to FT'$		$\{\ ?\ \}$
$\mid$	T_1/F	$\{\ T_0.v := T_1.v/F.v\ \}$	$T' \to *FT'$		$\{\ ?\ \}$
$\mid$	F	$\{\ T_0.v := F.v\ \}$	$\mid$	$/FT'$	$\{\ ?\ \}$
$F \to N$		$\{\ F.v := N.v\ \}$	$\mid$	ϵ	$\{\ ?\ \}$
$\mid$	(E)	$\{\ F.v := E.v\ \}$	$F \to N$		$\{\ ?\ \}$
			$\mid$	(E)	$\{\ ?\ \}$

Fig. 3. Simple expression definition (left) with left recursion removed (right)

so that, in our example, expressions are calculated correctly. The left recursive attributed grammar for expressions is given in on the left side of Figure 3. The right side shows the context-free grammar with left recursion removed. It is not obvious at a first glance, how the semantic rules have to be modified to describe the same meaning. To see how the semantic rules have to be modified, we examine the computation for the expression `1+2*3`. Figure 4 depicts the constructed abstract syntax tree together with the computation of the attributes, using the original (left) and transformed (right) grammar. As can be seen on the tree from the transformed grammar, the original tree has been stretched. The given evaluation for the transformed tree presents the basic idea: The computation of synthesized attributes is redirected to inherited attributes (i) of newly introduced nonterminals. From the leaves, the

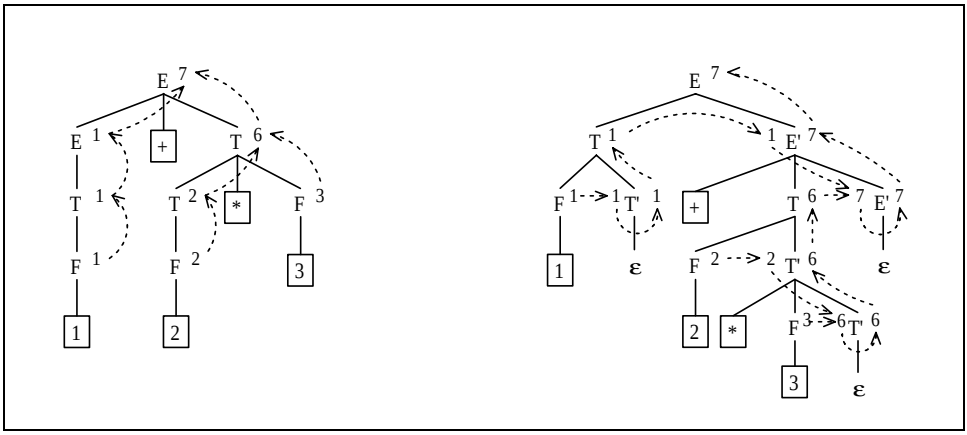

Fig. 4. Attributed syntax tree for `1+2*3` (left) and without left recursion (right)

results are copied using synthesized attributes. The intermediate results thus are preserved and combined to the final value with different positions only. The new semantic rules to achieve such behaviour are given in Figure 5.

$$
\begin{array}{rcll}
E & \rightarrow & TE' & \{\ E'.i := T.v, E.v := E'.v\ \} \\
E'_0 & \rightarrow & +TE'_1 & \{\ E'_1.i := E'_0.i + T.v, E'_0.v := E'_1.v\ \} \\
 & | & -TE'_1 & \{\ E'_1.i := E'_0.i - T.v, E'_0.v := E'_1.v\ \} \\
 & | & \epsilon & \{\ E'.v := E'.i\ \} \\
T & \rightarrow & FT' & \{\ T'.i := F.v, T.v := T'.v\ \} \\
T'_0 & \rightarrow & *FT'_1 & \{\ T'_1.i := T'_0.i * F.v, T'_0.v := T'_1.v\ \} \\
 & | & /FT'_1 & \{\ T'_1.i := T'_0.i/F.v, T'_0.v := T'_1.v\ \} \\
 & | & \epsilon & \{\ T'.v := T'.i\ \} \\
F & \rightarrow & N & \{\ F.v := N.v\ \} \\
 & | & (E) & \{\ F.v := E.v\ \}
\end{array}
$$

Fig. 5. Expression with removed left recursion and migrated semantic rules

4 General transformation algorithm for S-AG

The section gives the algorithm for left recursion removal for S-attribute grammars. The approach to migrate semantic rules guarantees that the root of the transformed syntax tree contains the same attribute values as the root in the original tree. A justification will be given in the next section.

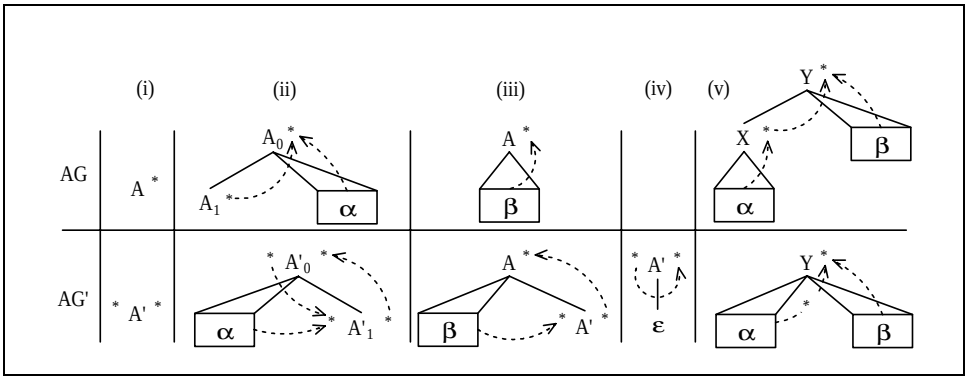

Fig. 6. Transformations of an S-attribute grammar, * representing inherited (on left side) and synthesized (on right side) attributes of a nonterminal

The algorithm for left recursion removal will be extended as follows (cf. Fig. 6):

(i) for each nonterminal A' newly introduced during transformation holds:

$$A_S(A') = A_S(A)$$

(1) $$A_I(A') = \{a' \mid a \in A_S(A)\} \quad \text{with} \quad TYPE(a') = TYPE(a)$$

A' gets all attributes of A, additionally an inherited attribute with the same type for each synthesized attribute of A.

(ii) During transformation of production $p : A_0 \to A_1\alpha$ to $p' : A'_0 \to \alpha A'_1$

$$R(p) = \{A_0.a := f_a(X_1^p.a_1, \ldots, X_{n_a}^p.a_{n_a}) \mid a \in A_S(A)\} \quad \Longrightarrow$$
$$R(p') = \{A'_1.a' := f_a(X_1^{p'}.a_1, \ldots, X_{n_a}^{p'}.a_{n_a}) \mid a' \in A_I(A')\} \cup$$
(2) $$\{A'_0.a := A'_1.a \mid a \in A_S(A)\}$$

$$\text{where} \quad X_i^{p'}.a_i = \begin{cases} A'_0.a'_i, \text{ if } X_i^p = A_1 \\ X_i^p.a_i, \text{ otherwise} \end{cases}$$

The actual computation is redirected to the inherited attributes. For synthesized attributes new copy rules are added.

(iii) For translation of a production $p : A^p \to \beta$ to $p' : A^{p'} \to \beta A'$

$$R(p) = \{A^p.a := f_a(X_1.a_1, \ldots, X_{n_a}.a_{n_a}) \mid a \in A_S(A)\} \quad \Longrightarrow$$
$$R(p') = \{A'.a' := f_a(X_1.a_1, \ldots, X_{n_a}.a_{n_a}) \mid a' \in A_I(A)\} \cup$$
(3) $$\{A^{p'}.a := A'.a \mid a \in A_S(A)\}$$

Similar, but without replacements of parameters for semantic rules.

(iv) Adding a new production $p : A' \to \epsilon$ requires

(4) $$R(p) = \{A'.a := A'.a' \mid a \in A_S(A')\}.$$

Copy rules are added from each inherited attribute to the corresponding

synthesized attribute.

(v) During transition of a production $p : Y \to X\beta$ to $p' : Y \to \alpha\beta$ by deploying $q : X \to \alpha$ with

$$\begin{aligned} R(q) &= \{X^q.a := f_a^q(\ldots) \mid a \in A_S(X)\} \\ R(p) &= \{Y^p.a := f_a^p(X_1^p.a_1, \ldots, X_{n_a}^p.a_{n_a}) \mid a \in A_S(Y)\} \qquad \Longrightarrow \\ R(p') &= \{Y^{p'}.a := f_a^p(X_1^{p'}.a_1, \ldots, X_{n_a}^{p'}.a_{n_a}) \mid a \in A_S(Y)\} \end{aligned}$$

$$\text{where } X_i^{p'}.a_i = \begin{cases} f_{a_i}^q(\ldots), \text{ if } X_i^p = X \\ X_i^p.a_i, \quad \text{otherwise} \end{cases} \tag{5}$$

Deploying the right hand side of a context-free rule the corresponding right hand side of a semantic rule is deployed parallely. As a consequence, $Y^{p'}.a$ is computed by a nested function application, which is not in line with the form given in Section 2. (The nested function application could be folded into semantic rules for the appropriate attribute to remove it.)

To sum up, the algorithm describes the transformation $AG \mapsto AG'$ of an S-attributed grammar $AG = (G, SD, AD, R)$ into $AG' = (G', SD, AD', R')$ with $G \mapsto G'$ according to the general algorithm for left recursion removal, $AD \mapsto AD'$ (1), and $R \mapsto R'$ (2 - 5).

5 Preservation of computed attribute values

Proposition: *For each transformation $AG \mapsto AG'$ following Section 4 holds: For each word derivable from the context-free grammar of AG and AG' all attribute occurrences in the root nodes of the corresponding syntax trees have the same values.*

Moreover, intermediate results are preserved in case of direct left recursion removal, though at different positions in the tree than in the original one.

In general, a left recursive rule is of the form $A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$. The choice of α_i and β_j does not matter for the argumentation, hence we assume $A \to A\alpha \mid \beta$ (with $\alpha, \beta \in V^*$). It can be seen that each such rule generates symbol sequences of the form $\beta\alpha^n$ (cf. for example, [18]), similarly to the corresponding transformed rules $A \to \beta A'$ and $A' \to \alpha A' \mid \epsilon$. Fig. 7 shows syntax trees for the derivation $\beta\alpha^n$. The α_i represent different derivations possible from α (instead of different alternatives of the rule). Note, that in general α_i and β can contain subtrees created by application of left recursive rules. Thus, they would need to be transformed into α_i^T and β^T.

We denote root nodes in the highest level of α (all roots of the forests) by R_α, $R_\alpha.a$ denotes an attribute occurrence at one of these nodes. A_0^T de-

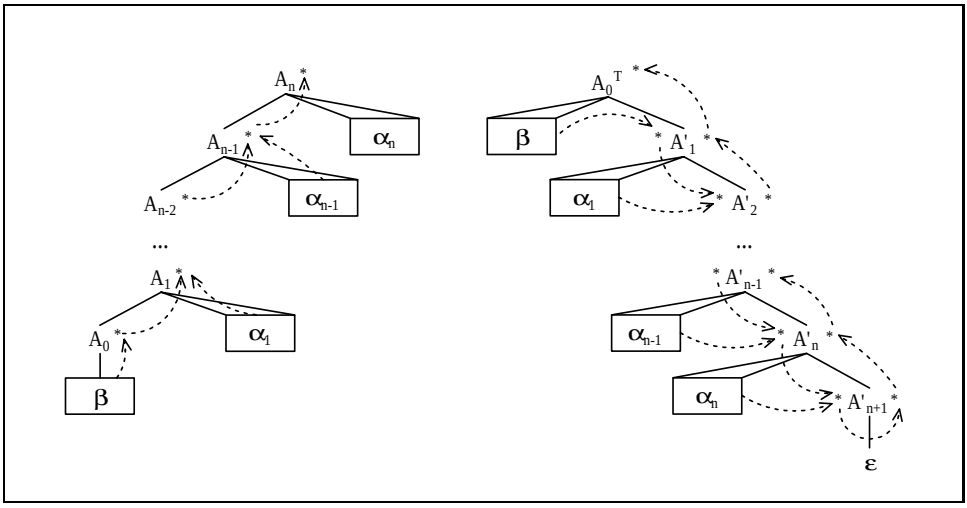

Fig. 7. Syntax trees for the derivation of $\beta\alpha^n$

notes the nonterminal A at the root of the derivation tree of the transformed production.

Precondition (using structural induction on trees) is

$$\forall a \in A_S(R_{\alpha_i}) : R_{\alpha_i}.a = R_{\alpha_i^T}.a, \quad \forall a \in A_S(R_\beta) : R_\beta.a = R_{\beta^T}.a \tag{6}$$

Base case is the largest left recursive subtree without left recursive subtrees. (6) holds, because $\alpha_i = \alpha_i^T, \beta = \beta^T$. Hence, we will not distinguish between α_i and α_i^T as well as β and β^T.

For the removal of direct left recursion the induction step is to show that for the transformation depicted in Fig. 7 holds

$$\forall a \in A_S(A) : A_n.a = A_0^T.a \tag{7}$$

Therefore, we need the equation

$$\forall i \in \{0, \ldots, n\} \forall a \in A_S(A) : A_i.a = A'_{i+1}.a' \tag{8}$$

To show that it is valid, we use induction over depth of derivation trees n: Base case ($n = 0$):

$$\begin{aligned} A_0.a &= f(R_\beta.a_1, \ldots, R_\beta.a_{n_a}) && \text{cf. Def.} \\ A'_1.a' &= f(R_\beta.a_1, \ldots, R_\beta.a_{n_a}) && \text{cf. (3)} \\ \hookrightarrow \quad A_0.a &= A'_1.a' \qquad \forall a \in A_S(A) \end{aligned}$$

Induction step $(n \mapsto n+1)$:

$$\begin{aligned}
A_{n+1}.a &= f(A_n.a_1, \ldots, A_n.a_{n_a}, R_{\alpha_{n+1}}.a_1, \ldots, R_{\alpha_{n+1}}.a_m) && \text{cf. Def.}\\
A'_{n+2}.a' &= f(A'_{n+1}.a'_1, \ldots, A'_{n+1}.a'_{n_a}, R_{\alpha_{n+1}}.a_1, \ldots, R_{\alpha_{n+1}}.a_m) && \text{cf. (2)}\\
A_i.a_j &= A'_{i+1}.a'_j \qquad \forall j \in \{1, \ldots, n_a\} && \text{ind.assp.}\\
\hookrightarrow \quad A_{n+1}.a &= A'_{n+2}.a' \qquad \forall a \in A_S(A)
\end{aligned}$$

Thus, (8) holds for all n, and we can say that intermediate results of computations are preserved in computations of inherited attributes of other, well defined nodes in the transformed tree. Now holds $\forall a \in A_S(A)$:

$$\begin{aligned}
\forall i, j \in \{1, \ldots, n+1\} : \; A'_i.a &= A'_j.a && \text{conclusion from (2)}\\
A_0^T.a &= A'_1.a && \text{cf. (3)}\\
\hookrightarrow \qquad A_0^T.a &= A'_{n+1}.a\\
&= A'_{n+1}.a' && \text{cf. (4)}\\
&= A_n.a && \text{cf. (8)}
\end{aligned}$$

From (5) we can conclude that attribute values in the root do not change by deployment of the right hand side of a rule while removing indirect left recursion.

6 Non-S-attributed grammars

I-AGs can be treated nearly analogously. Attributes are computed and copied top down. Intuitively, the procedure is turned upside down, i.e. the actual computation is done on newly added synthesized attributes and are copied downwards using existing inherited attributes. In simple multi-pass AGs, each attribute can be computed during a certain pass. We can suppose that each pass is defined by an S-AG or an I-AG. Therefore, our approach can be generalised to simple multi-pass AGs. Because a multi-pass AG can be transformed into an equivalent simple multi-pass AG such AGs can be treated, too.

7 Practical experiences

Prototype The given approach has been implemented as proof-of-concept prototype for multi-pass attribute grammar. I.e. it demonstrates the algorithm for simple examples, but is not ready for practical applications. For the implementation TXL [2] was chosen. For the experiments, we used grammars as

```
e(V) :- e(E), @"+", t(T), V is E+T.
e(V) :- e(E), @"-", t(T), V is E-T.
e(V) :- t(T), V is T.

t(V) :- t(T), @"*", f(F), V is T*F.
t(V) :- t(T), @"/", f(F), V is T/F.
t(V) :- f(F), V is F.

f(V) :- num(N), V is N.
f(V) :- @"(", e(E), @")", V is E.
```

```
TXL v10.3 (8.3.03) (c)1988-2003 Queen's University
Compiling ag.Txl ...
Parsing samples/exp.ag ...
Transforming ...
e1(V,V3) :- @"+", t(T), e1(V,E2), E2 is V3+T.
e1(V,V4) :- @"-", t(T), e1(V,E3), E3 is V4-T.
e(V) :- t(T), e1(V,V2), V2 is T.
t1(V,V7) :- @"*", f(F), t1(V,T2), T2 is V7*F.
t1(V,V8) :- @"/", f(F), t1(V,T3), T3 is V8/F.
t(V) :- f(F), t1(V,V6), V6 is F.
f(V) :- num(N), V is N.
f(V) :- @"(", e(E), @")", V is E.
e1(V,V).
t1(V,V).
```

Fig. 8. Input (left) and output (right) of the prototype

used by Laptob [14]. Grammars are represented as logic rules in Prolog. Predicates represent nonterminals, `@` interprets strings as terminals, and variables are attributes addressed by position. In Figure 8 the input (left) and output (right) for the prototype is given. The prototype is under reimplementation in Prolog for a smoother integration with Laptob.

A larger scenario We started to apply the approach to a 15 years evolutionary grown YACC specification [1] describing LPC, a language for interpreted scripts in a multi-user environment [2]. The grammar currently possesses 99 rules with 310 alternatives altogether, and it is likely to change in future. We have no influence on grammar and code, as this is part of a kernel distribution for 100s of such environments. Since more than one year, complex modernisations of the class library are being done. As a consequence, there are 1000s of changes in the area code. Tool support is desirable, where each necessary change is specified with semantic rules according to the known grammar. For several reasons, an LL(k) grammar based tool was chosen. Figure 9 shows an extract from the context-free grammar of the definition for the expression. Even without left recursion removal, real grown grammar rules are difficult to read. The grammar rules can be automatically extracted from the YACC specification and converted in the grammar notation used for the tool. The context-free part of the grammar is then reused to specify source-to-source transformations by giving appropriate semantic rules. The above approach can then be used to transform the transformation into a form suitable for the used tool, as is demonstrated in Figure 10. Several technical problems have still to be solved. For example, ϵ-productions violate the conditions to apply the algorithm for left recursion removal.

[1] http://www.ldmud.de
[2] http://www.evermore.org

```
expr0 :                          (some of 34 alternatives)
    lvalue L_ASSIGN expr0 %prec L_ASSIGN
    | expr0 '?' expr0 ':' expr0 %prec '?'
    | expr0 L_LOR %prec L_LOR expr0
    | expr0 '|' expr0       | decl_cast expr0 %prec '~'   | cast expr0 %prec '~'
    | pre_inc_dec expr4 index_expr %prec '['
    | pre_inc_dec expr4 '[' expr0 ',' expr0 ']' %prec '['
    | L_NOT expr0 | '-' expr0 %prec '~'
    | expr4
    ...
expr4 :                              (some of 29 alternatives)
    | inline_func     |  catch    |  L_CLOSURE | L_SYMBOL | L_FLOAT
    | '(' note_start comma_expr ')'          | '(' '{' note_start expr_list '}' ')'
    | L_QUOTED_AGGREGATE note_start expr_list '}'  ')'
    | '(' '[' ':' expr0 ']' ')'   | '(' '[' m_expr_list ']' ')'   | '(' '<' '>' ')'
    | '(' '<' identifier '>' note_start opt_struct_init ')'
    | expr4 L_ARROW struct_member_name       | '&' '(' expr4 L_ARROW struct_member_name ')'
    | expr4 index_range
    | '&' L_LOCAL                           | '&' '(' expr4 index_expr ')'
    | '&' '(' expr4 '[' expr0 ',' expr0 ']' ')'| '&' '(' expr4 index_range ')'
    | expr4 index_expr   | expr4 '[' expr0 ',' expr0 ']' | L_LOCAL
    ...
```

Fig. 9. Context-free extract from a yacc specification for expression definition

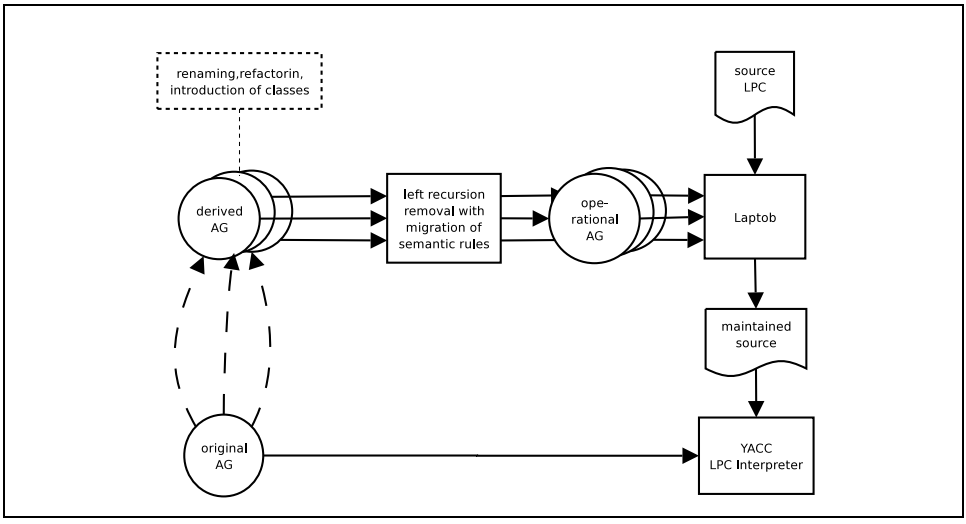

Fig. 10. Reuse of the original grammar for small maintenance transformations

8 Related Work

There are several approaches to left recursion removal, all of them dealing with the context-free grammar only, without caring for attributes. The general algorithm for left recursion removal is given in many compiler books, as representative see Louden [18]. He also demonstrates, how left recursion can

be avoided using EBNF-notation, and an implementation using iteration is given. Rechenberg/Mössenböck [21] use a translation of the grammar to syntax graphs, from which they construct parsers. Left recursion is handled by transforming it into iteration, while preserving the accepted language.

We mentioned the use of top-down tools for their ease of use. Pepper [20] unifies the paradigms for LR(k)- and LL(k)-parsing expressed by the formula LR(k) = 3NF + LL(k). The main aim is an easy comprehensible derivation method, easy to adapt, providing the power of LR parsing while providing efficiency known from LALR parsing. Grammars are enriched with null nonterminals, which do not change the language but may carry semantic actions or can act as assertions that guide reductions. Semantic rules are not considered during the grammar transformation process. Schmeiser/Barnard [22] modify the standard table driven algorithm for bottom-up parsing to offer the programmer a top-down parse order while using a bottom-up parser. Besides states additionally rule lists are stored on the stack. When a rule is reduced, the rule lists are concatenated in suitable order.

We discussed that grammars are not only changed to implement compilers. The need of an engineering discipline for grammarware is emphasised in [8]. In [15] the authors propose an approach to the construction of grammars for existing languages. The main characteristic of the approach is that the grammars are not constructed from scratch but they are rather recovered by extracting them from language references, compilers, and other artifacts. They provide a structured process to recover grammars including the automated transformation of raw extracted grammars and the derivation of parsers. Examples for tool support for grammar engineering are Grammar Deployment Kit (GDK) [7] and *F*ramework for *S*DF *T*ransformation (FST) [16]. GDK provides support in the process to turn a grammar specification into a working parser. FST supports the adaptation of grammars based on the syntax definition formalism SDF, where, for example, EBNF patterns are removed (YACCification) or introduced (deYACCification). Transformations by Cordy et al. to enable agile parsing based on problem-specific grammars [3,4] are examples for grammar engineering as well as the transformations for deriving an abstract from a concrete syntax by Wile [23].

Theoretical work on general grammar adaptations can be found in [11]. A set of operators is defined together with properties. The operators can be used to describe grammar adaptations.

Lämmel et al. [12,13,10] also work on grammar evolution. For example, a general framework for meta-programming is developed in [10] together with an operator suite, where its operators model schemata of program transformation, synthesis and composition. Examples are fold and unfold operations defined

on skeletons of declarative programs, i.e. attribute grammars, logic programs. Parameters are analysed and propagated through folded elements.

There is also a relation to refactoring [5,19], which can be applied to grammars and transformation rules. Indeed, left recursion removal could be considered as a composite refactoring for grammars.

Related to this paper is a former paper on automatic migration of transformation rules after a grammar extension has been made [17]. The approach can be used to reuse transformation rules after a grammar extension, so that they do not break with code for the new grammar. It was shown on the problem, how to make rewrite rules able to store layout information in the rewrite pattern. On the level for the rewriter, that information was invisible, thus the approach helped to reduce complexity of rewrite patterns for users.

9 Concluding remarks

Summary The paper contributes to the work on grammar adaptations and concentrates on semantics rules associated to grammar productions. The approach attempts to reuse existing semantic rules for the new grammar. Moreover, it offers the programmer of a program transformation the opportunity to specify semantic rules on a grammar closer to a grammar specification, while grammar and semantic rules can be adapted to meet technical demands, here left recursion removal. Hence, we provide the rewriter with a simpler grammar than necessary for the tool. The necessary transformation steps for the grammar are given, as well as a justification of the approach.

A disadvantage of the approach is the doubling of attribute numbers, and the introduction of additional copy rules. Though the added complexity is hidden, the problem might be the time overhead it adds to the process of tool construction. Semantic rules using the original grammar have to be adapted each time a tool is built from rules and grammar. In the case of interpretative used environments, e.g. in a Prolog setting, this may become annoying for larger grammars.

Future Work We are going to make the approach real life usable, the current state is still a weak prototype. There is still the problem of ϵ-productions. The algorithm could benefit from improvement by the use of lazy evaluation strategies. The approach would then only be implemented with S-AG, all other variants are automatically supported. It is also possible to construct terms instead of applying operations, then interpret the term in the root attributes.

In future we will look for further grammar adaptations necessary during maintenance and investigate, if and how it is possible to derive changes for both, the software the grammar uses and for the semantic rules associated

with the grammar. We will examine, how we can connect such combined grammar/ transformation rule adaptations to more complex operations.

Acknowledgement

We would like to thank the anonymous reviewers for their suggestions and Anke Dittmar for their remarks on an earlier draft of the paper. We are grateful for collaboration with Ralf Lämmel on the subject of grammar engineering and meta-programming. The idea to improve the algorithm by using a lazy evaluation strategy has been pointed out by Anke Dittmar. Jim Cordy helped us in questions concerning TXL.

References

[1] Alblas, H., *Introduction to Attribute Grammars*, in: H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems (SAGA 1991)*, LNCS **545** (1991), pp. 1–15.

[2] Cordy, J., T. Dean, A. Malton and K. Schneider, *Source Transformation in Software Engineering using the TXL T ransformation System*, Journal of Information and Software Technology **44** (2002), pp. 827–837.

[3] Dean, T., J. Cordy, A. Malton and K. Schneider, *Grammar Programming in TXL*, in: *Proc. Source Code Analysis and Manipulation (SCAM'02)* (2002).

[4] Dean, T., J. Cordy, A. Malton and K. Schneider, *Agile Parsing in TXL*, Journal of Automated Software Engineering **10** (2003), pp. 311–336.

[5] Fowler, M., K. Beck, J. Brant, W. Opdyke and D. Roberts, "Refactoring: Improving the Design of Existing Code," Addison-Wesley, 1999.

[6] Irons, E. T., *A syntax-directed compiler for ALGOL 60*, Communications of the ACM **4** (1961), pp. 51–55.

[7] Jan Kort and Ralf Lämmel and Chris Verhoef, *The Grammar Deployment Kit*, in: M. G. v. d. Brand and R. Lämmel, editors, *Electronic Notes in Theoretical Computer Science*, ENTCS **65** (2002).

[8] Klint, P., R. Lämmel and C. Verhoef, *Towards an engineering discipline for grammarware* (2003), 32 pages, submitted for journal publication.

[9] Knuth, D. E., *Semantics of context-free languages*, Mathematical Systems Theory **2** (1968), pp. 127–145.

[10] Lämmel, R., "Functional meta-programs towards reusability in the declarative paradigm," Ph.D. thesis, University of Rostock, Department of Computer Science (1999).

[11] Lämmel, R., *Grammar Adaptation*, in: *Proc. Formal Methods Europe (FME) 2001*, LNCS **2021** (2001), pp. 550–570.

[12] Lämmel, R., *Evolution of Rule-Based Programs*, Journal of Logic and Algebraic Programming (2004), Special Issue on Structural Operational Semantics; To appear.

[13] Lämmel, R. and G. Riedewald, *Reconstruction of paradigm shifts*, in: *Second Workshop on Attribute Grammars and their Applications, WAGA 99*, 1999, pp. 37–56, INRIA, ISBN 2-7261-1138-6.

[14] Lämmel, R. and G. Riedewald, *Prological Language Processing*, in: M. G. v. d. Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, ENTCS **44** (2001). URL http://www.elsevier.com/locate/entcs/volume44.html

[15] Lämmel, R. and C. Verhoef, *Semi-automatic Grammar Recovery*, Software—Practice & Experience **31** (2001), pp. 1395–1438.

[16] Lämmel, R. and G. Wachsmuth, *Transformation of SDF syntax definitions in the ASF+SDF Meta-Environment*, in: M. van den Brand and D. Parigot, editors, *Proceedings of the First Workshop on Language Descriptions, Tools and Applications (LDTA'01), Genova, Italy, April 7, 2001, Satellite event of ETAPS'2001*, ENTCS **44** (2001).

[17] Lohmann, W. and G. Riedewald, *Towards automatical migration of transformation rules after grammar extension*, in: *Proc. of 7th European Conference on Software Maintenance and Reengineering (CSMR'03)* (2003), pp. 30–39.

[18] Louden, K. C., "Compiler construction: principles and practice," International Thomson Publishing, 1997.

[19] Opdyke, W. F. and R. J. Johnson, *Refactoring: An Aid in Designing Application Frameworks*, in: *Proceedings of the Symposium on Object-Oriented Programming emphasizing Practical Applications*, ACM-SIGPLAN, 1990, pp. 145–160.

[20] Pepper, P., *LR Parsing = Grammar Transformation + LL Parsing*, Technical Report CS-99-05, TU Berlin (1999).

[21] Rechenberg, P. and H. Mössenböck, "Ein Compiler - Generator für Mikrocomputer," Carl Hanser Verlag, 1985, in german.

[22] Schmeiser, J. P. and D. T. Barnard, *Producing a top-down parse order with bottom-up parsing*, Information Processing Letters **54** (1995), pp. 323–326.

[23] Wile, D., *Abstract syntax from concrete syntax*, in: *Proc. International Conference on Software Engineering (ICSE'97)* (1997), pp. 472–480.